

## **1.FOLDER: LOADER\_FROM\_DIRECTORY.PY :IMPLIES THE FOLLOWING**

*CODE: # to measure run-time*

```
import time
```

```
# to shuffle data
```

```
import random
```

```
# for plotting
```

```
import matplotlib.pyplot as plt
```

```
# pytorch stuff
```

```
import torch
```

```
import torch.nn as nn
```

```
import torch.optim as optim
```

```
# for csv dataset
```

```
import torchvision
```

```
import csv
```

```
import os
```

```
import pandas as pd
```

```
from urllib import request
```

```
# import statements for iterating over csv file
```

```
import cv2
```

```
import numpy as np
```

```
import urllib.request
```

```
# to get the alphabet
```

```
import string
```

```
from PIL import Image
```

```

# generate the targets

# the targets are one hot encoding vectors

alphabet = list(string.ascii_lowercase)

target = {}

# Initialize a target dict that has letters as its keys and empty one-hot encoding vectors of size 37 as its values

for letter in alphabet:
    target[letter] = [0] * 37

# Do the one-hot encoding for each letter now

curr_pos = 0

for curr_letter in target.keys():
    target[curr_letter][curr_pos] = 1
    curr_pos += 1

# extra symbols

symbols = ["space", "number", "period", "comma", "colon", "apostrophe", "hyphen", "semicolon",
"question", "exclamation", "capitalize"]

# create vectors

for curr_symbol in symbols:
    target[curr_symbol] = [0] * 37

# create one-hot encoding vectors

for curr_symbol in symbols:
    target[curr_symbol][curr_pos] = 1
    curr_pos += 1

# collect all data from the csv file

data = []

```

```

for tgt in os.listdir("dataset"):

    if not tgt == ".DS_Store":

        for folder in os.listdir("dataset/" + tgt + "/Uploaded"):

            if not folder == ".DS_Store":

                for filename in os.listdir("dataset/" + tgt + "/Uploaded/" + folder):

                    if not filename == ".DS_Store":


                        # store the image and label

                        picture = []

                        curr_target = target[tgt]

                        image = Image.open("dataset/" + tgt + "/Uploaded/" + folder + "/" + filename)

                        image = image.convert('RGB')

                        # f.show()

                        image = np.array(image)

                        # resize image to 28x28x3

                        image = cv2.resize(image, (28, 28))

                        # normalize to 0-1

                        image = image.astype(np.float32)/255.0

                        image = torch.from_numpy(image)

                        picture.append(image)

                        # convert the target to a long tensor

                        curr_target = torch.Tensor([curr_target])

                        picture.append(curr_target)

                        # append the current image & target

                        data.append(picture)

# create a dictionary of all the characters

characters = alphabet + symbols

index2char = {}

number = 0

```

```
for char in characters:
    index2char[number] = char
    number += 1

# find the number of each character in a dataset
def num_chars(dataset, index2char):
    chars = {}
    for _, label in dataset:
        char = index2char[int(torch.argmax(label))]
        # update
        if char in chars:
            chars[char] += 1
        # initialize
        else:
            chars[char] = 1
    return chars

# Create dataloader objects

# shuffle all the data
random.shuffle(data)

# batch sizes for train, test, and validation
batch_size_train = 30
batch_size_test = 30
batch_size_validation = 30

# splitting data to get training, test, and validation sets
# change once get more data
# 1600 for train
train_dataset = data[:22000]
```

```

# test has 212
test_dataset = data[22000:24400]

# validation has 212
validation_dataset = data[24400:]

# create the dataloader objects

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size_train,
shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size_test,
shuffle=False)

validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset,
batch_size=batch_size_validation, shuffle=True)

# to check if a dataset is missing a char
test_chars = num_chars(test_dataset, index2char)

num = 0

for char in characters:
    if char in test_chars:
        num += 1
    else:
        break
print(num)

class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.block1 = nn.Sequential(
            #3x28x28
            nn.Conv2d(in_channels=3,
                     out_channels=16,
                     kernel_size=5,

```

```
        stride=1,  
        padding=2),  
        # batch normalization  
        # nn.BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True),  
        #16x28x28  
        nn.MaxPool2d(kernel_size=2),  
        #16x14x14  
        nn.LeakyReLU()  
    )  
    #16x14x14  
    self.block2 = nn.Sequential(  
        nn.Conv2d(in_channels=16,  
                 out_channels=32,  
                 kernel_size=5,  
                 stride=1,  
                 padding=2),  
        # batch normalization  
        # nn.BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True),  
        #32x14x14  
        nn.MaxPool2d(kernel_size=2),  
        #32x7x7  
        nn.LeakyReLU()  
    )  
    # linearly  
    self.block3 = nn.Sequential(  
        nn.Linear(32*7*7, 100),  
        # batch normalization  
        # nn.BatchNorm1d(100),  
        nn.LeakyReLU(),  
        nn.Linear(100, 37)  
    )
```

#1x37

```
def forward(self, x):
    out = self.block1(x)
    out = self.block2(out)
    # flatten the dataset
    out = out.view(-1, 32*7*7)
    out = self.block3(out)

    return out

# convolutional neural network model
model = CNN()

# print summary of the neural network model to check if everything is fine.
print(model)
print("# parameter: ", sum([param.nelement() for param in model.parameters()]))

# setting the learning rate
learning_rate = 1e-4

# Using a variable to store the cross entropy method
criterion = nn.CrossEntropyLoss()

# Using a variable to store the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# list of all train_losses
train_losses = []

# list of all validation losses
```

```
validation_losses = []

# for loop that iterates over all the epochs
num_epochs = 10

for epoch in range(num_epochs):

    # variables to store/keep track of the loss and number of iterations
    train_loss = 0
    num_iter_train = 0

    # train the model
    model.train()

    # Iterate over train_loader
    for i, (images, labels) in enumerate(train_loader):
        # need to permute so that the images are of size 3x28x28
        # essential to be able to feed images into the model
        images = images.permute(0, 3, 1, 2)

        # Zero the gradient buffer
        # resets the gradient after each epoch so that the gradients don't add up
        optimizer.zero_grad()

        # Forward, get output
        outputs = model(images)

        # convert the labels from one hot encoding vectors into integer values
        labels = labels.view(-1, 37)
        y_true = torch.argmax(labels, 1)

        # calculate training loss
```

```
loss = criterion(outputs, y_true)

# Backward (computes all the gradients)
loss.backward()

# Optimize
# loops through all parameters and updates weights by using the gradients
# takes steps backwards to optimize (to reach the minimum weight)
optimizer.step()
# update the training loss and number of iterations
train_loss += loss.data[0]
num_iter_train += 1

print('Epoch: {}'.format(epoch+1))
print('Training Loss: {:.4f}'.format(train_loss/num_iter_train))
# append training loss over all the epochs
train_losses.append(train_loss/num_iter_train)

# evaluate the model
model.eval()

# variables to store/keep track of the loss and number of iterations
validation_loss = 0
num_iter_validation = 0

# Iterate over validation_loader
for i, (images, labels) in enumerate(validation_loader):
    # need to permute so that the images are of size 3x28x28
    # essential to be able to feed images into the model
    images = images.permute(0, 3, 1, 2)
```

```
# Forward, get output
outputs = model(images)

# convert the labels from one hot encoding vectors to integer values
labels = labels.view(-1, 37)
y_true = torch.argmax(labels, 1)

# calculate the validation loss
loss = criterion(outputs, y_true)

# update the training loss and number of iterations
validation_loss += loss.data[0]
num_iter_validation += 1

print('Validation Loss: {:.4f}'.format(validation_loss/num_iter_validation))

# append all validation_losses over all the epochs
validation_losses.append(validation_loss/num_iter_validation)

num_iter_test = 0
correct = 0

# Iterate over test_loader
for images, labels in test_loader:

    # need to permute so that the images are of size 3x28x28
    # essential to be able to feed images into the model
    images = images.permute(0, 3, 1, 2)

    # Forward
    outputs = model(images)
```

```

# convert the labels from one hot encoding vectors into integer values
labels = labels.view(-1, 37)

y_true = torch.argmax(labels, 1)

# find the index of the prediction
y_pred = torch.argmax(outputs, 1).type('torch.FloatTensor')

# convert to FloatTensor
y_true = y_true.type('torch.FloatTensor')

# find the mean difference of the comparisons
correct += torch.sum(torch.eq(y_true, y_pred).type('torch.FloatTensor'))

print('Accuracy on the test set: {:.4f}%'.format(correct/len(test_dataset) * 100))
print()

# learning curve function

def plot_learning_curve(train_losses, validation_losses):
    # plot the training and validation losses
    plt.ylabel('Loss')
    plt.xlabel('Number of Epochs')
    plt.plot(train_losses, label="training")
    plt.plot(validation_losses, label="validation")
    plt.legend(loc=1)

# plot the learning curve
plt.title("Learning Curve (Loss vs Number of Epochs)")
plot_learning_curve(train_losses, validation_losses)

torch.save(model.state_dict(), "model.pth")

```

## **DESCRIBES THE FOLLOWING:**

Hey Nandana! 🌟 This code is a **complete pipeline** for **Braille character recognition** using a **Convolutional Neural Network (CNN)**. It processes a labeled image dataset of Braille characters, trains a CNN model on it, and evaluates its performance. Let's break it down **step-by-step** in a human-understandable and descriptive way.

---

### **Goal**

Recognize 37 classes:

- 26 lowercase English letters (a-z)
  - 11 special symbols like space, number, period, etc.
- 

### **Libraries & Imports**

The code imports everything needed for:

- **Data handling:** pandas, csv, os, urllib, string
  - **Image processing:** PIL, cv2, numpy
  - **Deep Learning:** torch, torchvision, nn, optim
  - **Visualization:** matplotlib.pyplot
  - **Timing & Randomization:** time, random
- 

### **One-Hot Encoding of Labels**

It creates **one-hot vectors** of size 37 for each character (a-z + symbols).

```
target['a'] = [1, 0, 0, ..., 0] # 37 elements
```

```
target['space'] = [0, 0, ..., 1]
```

---

### **Dataset Preparation**

- Reads image files from dataset/{char}/Uploaded/{subfolder}/filename.jpg
  - Converts each image to RGB, resizes it to **28x28**, and normalizes pixel values to **[0,1]**
  - Associates it with the correct **one-hot encoded target**
  - Data format: [[image\_tensor, label\_tensor], ...]
- 

### **Data Splitting**

*Splits the dataset into:*

- **Training:** 22000 samples
- **Testing:** 2400 samples
- **Validation:** rest of the data

*Creates corresponding DataLoader objects to handle batching and shuffling.*

---

### Check Missing Classes

*Counts how many of the 37 classes are represented in the test set.*

---

### CNN Architecture

*A custom CNN (CNN class) with the following layers:*

#### 1. **Block 1**

- Conv2D ( $3 \rightarrow 16$  filters)
- MaxPooling (2x2)
- LeakyReLU

#### 2. **Block 2**

- Conv2D ( $16 \rightarrow 32$  filters)
- MaxPooling (2x2)
- LeakyReLU

#### 3. **Fully Connected**

- Linear ( $32 \times 7 \times 7 \rightarrow 100$ )
- LeakyReLU
- Linear ( $100 \rightarrow 37$  output classes)

 *This CNN learns features from the input images and predicts one of 37 characters.*

---

### Model Training

*For num\_epochs = 10, it:*

1. Feeds batches of images into the model (train\_loader)
2. Applies **cross-entropy loss** for multi-class classification
3. Performs **backpropagation** and **weight updates** using **Adam optimizer**
4. Stores average **training loss per epoch**

---

### Validation & Testing

- After each epoch, evaluates model on the validation set and calculates loss.
  - Also checks **accuracy on the test set**:
    - Compares predicted vs. true label
    - Measures how many characters were correctly identified
- 

### Plotting the Learning Curve

Shows how **training and validation loss** change over epochs using Matplotlib.

```
plt.plot(train_losses, label="training")  
plt.plot(validation_losses, label="validation")
```

This helps detect underfitting/overfitting visually.

---

### Save the Model

```
torch.save(model.state_dict(), "model.pth")
```

Saves the trained model parameters to a .pth file for future use.

---

### Output You'll See

After training, for each epoch:

- Training loss
- Validation loss
- Test accuracy in percentage

Example:

Epoch: 1

Training Loss: 0.5623

Validation Loss: 0.4891

Accuracy on the test set: 85.21%

---

### Summary

This code:

- Loads and preprocesses image data

- *Encodes labels*
  - *Defines a CNN model for character classification*
  - *Trains and validates the model*
  - *Evaluates accuracy*
  - *Plots loss curves*
  - *Saves the final trained model*
- 

## **2. THE FOLDER :SPACE\_RECOGNITION\_ORIGINAL.PY IMPLIES:**

**CODE:**

```
import torch
import torch.nn as nn
# import torch.onnx
# import onnx
# import onnx_caffe2.backend
#from onnx import checker, helper
import torch.optim as optim
import numpy as np
import cv2
from PIL import Image
import torch.utils.model_zoo as model_zoo
import torch.onnx

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.block1 = nn.Sequential(
            # 3x28x28
            nn.Conv2d(in_channels=3,
                      out_channels=16,
                      kernel_size=5,
                      stride=1,
```

```

    padding=2),
    # 16x28x28
    nn.MaxPool2d(kernel_size=2),
    # 16x14x14
    nn.LeakyReLU()
)
# 16x14x14
self.block2 = nn.Sequential(
    nn.Conv2d(in_channels=16,
        out_channels=32,
        kernel_size=5,
        stride=1,
        padding=2),
    # 32x14x14
    nn.MaxPool2d(kernel_size=2),
    # 32x7x7
    nn.LeakyReLU()
)
# linearly
self.block3 = nn.Sequential(
    nn.Linear(32 * 7 * 7, 100),
    nn.LeakyReLU(),
    nn.Linear(100, 37)
)
# 1x36

```

```

def forward(self, x):
    out = self.block1(x)
    out = self.block2(out)
    # flatten the dataset
    # ipdb; ipdb.set_trace()

```

```

out = out.view(-1, 32 * 7 * 7)
out = self.block3(out)

return out
#
# print(make_prediction("test/Prairie.jpg"))

model = CNN()
model.load_state_dict(torch.load(r"C:\\New folder\\Braille\\final_model\\model.pth"))

def export_model():
    model = CNN()
    model.load_state_dict(torch.load("model.pth"))

    # Input to the model
    x = torch.randn(5, 3, 28, 28)

    # Export the model
    torch_out = torch.onnx._export(model, # model being run
                                  x, # model input (or a tuple for multiple inputs)
                                  "model.onnx-2",
                                  # where to save the model (can be a file or file-like object)
                                  export_params=True) # store the trained parameter weights inside the model
file

def inspect_model():
    # Input image into the ONNX model
    onnx_model = onnx.load("model.onnx")
    model = onnx_caffe2.backend.prepare(onnx_model)

    image = Image.open("z.jpg")

```

```

# # image = image.convert('RGB')
image = np.array(image)
image = cv2.resize(image, (28, 28))
image = image.astype(np.float32) / 255.0
image = torch.from_numpy(image[None, :, :, :])
image = image.permute(0, 3, 1, 2)
W = {model.graph.input[0].name: image.data.numpy()}

model_out = model.run(W)[0]
print(model_out)

#
# # onnx_model(image)
#
# print(onnx_model)

# onnx.checker.check_model(onnx_model)
# # print(onnx.helper.printable_graph(onnx_model.graph))

def make_prediction(img_path):

    image = Image.open(img_path)
    image = image.convert('RGB')
    width, height = image.size
    num = round(width/height/0.78)
    w = width/num

    letters = []
    for i in range(0, num):
        cropped = image.crop((i * w, 0, (i + 1) * w, height))
        # cropped.show()
        cropped = np.array(cropped)
        cropped = cv2.resize(cropped, (28, 28))

```

```
cropped = cropped.astype(np.float32) / 255.0
cropped = torch.from_numpy(cropped[None, :, :, :])
cropped = cropped.permute(0, 3, 1, 2)
predicted_tensor = model(cropped)
_ , predicted_letter = torch.max(predicted_tensor, 1)
if int(predicted_letter) == 26:
        letters.append(chr(32))
elif int(predicted_letter) == 27:
        letters.append(chr(35))
elif int(predicted_letter) == 28:
        letters.append(chr(46))
elif int(predicted_letter) == 29:
        letters.append(chr(44))
elif int(predicted_letter) == 30:
        letters.append(chr(58))
elif int(predicted_letter) == 31:
        letters.append(chr(92))
elif int(predicted_letter) == 32:
        letters.append(chr(45))
elif int(predicted_letter) == 33:
        letters.append(chr(59))
elif int(predicted_letter) == 34:
        letters.append(chr(63))
elif int(predicted_letter) == 35:
        letters.append(chr(33))
elif int(predicted_letter) == 36:
        letters.append(chr(126))
else:
        letters.append(chr(97 + predicted_letter))

output = ""
```

```
number = False
capL = False
capW = False
for j in letters:
    if j == '#':
        number = True
    elif ord(j) == 126:
        if capL:
            capW = True
        capL = True
    elif j == ' ':
        number = False
        capL = False
        capW = False
        output = output + j
    elif not number:
        if capW and ord(j) in range(97, 123):
            output = output + chr(ord(j) - 32)
        elif capL and ord(j) in range(97, 123):
            output = output + chr(ord(j) - 32)
        capL = False
    else:
        output = output + j
else:
    if ord(j) in range(97, 106):
        output = output + chr(ord(j)-48)
    elif ord(j) == 106:
        output = output + chr(48)
    else:
        output = output + j
```

```

return output

# print(make_prediction(r"D:\Corezone\2024-2025\Checkings\aeye-alliance-master\aeye-alliance-
master\test\Shopify.png"))

# print(make_prediction("test/He_took"))
# print(make_prediction("test/the_little.png"))
# print(make_prediction("test/with_his_family.png"))
# print(make_prediction("test/with_his_mouth..png"))
# print(make_prediction("test/would_run_and_get_it.png"))

```

*DESCRIBES THE FOLLOWING:*

*Hey Nandana! 🤝 Here's a detailed and human-understandable breakdown of your Braille recognition project code, step-by-step. You're essentially building a **Braille-to-text converter using a CNN (Convolutional Neural Network)** trained to classify Braille characters from images.*

---

## 1. Library Imports

```

import torch, torch.nn as nn, torch.optim as optim

import numpy as np, cv2

from PIL import Image

import torch.onnx

```

*These are required for:*

- **Torch:** For model building, training, loading, and inference.
- **cv2 (OpenCV) & PIL:** For image processing.
- **NumPy:** For numerical operations.

- **ONNX (commented):** For exporting the model to an interoperable format.
- 

## 2. CNN Model Architecture

*class CNN(nn.Module):*

...

**This model processes 28x28 RGB images and classifies them into 37 classes:**

- **26 letters (a-z)**
- **11 symbols/special characters** like space, comma, colon, exclamation, etc.

**Structure:**

- **Block 1:**
  - Conv2D (3→16 channels) → MaxPooling → LeakyReLU
  - Output: 16 x 14 x 14
- **Block 2:**
  - Conv2D (16→32 channels) → MaxPooling → LeakyReLU
  - Output: 32 x 7 x 7
- **Block 3:**
  - Flatten to 1D
  - Fully connected: 32\*7\*7 → 100 → 37 (final output classes)

**Output:**

*Returns a vector of size 37 for classification.*

---

## 3. Model Loading

*model.load\_state\_dict(torch.load("path/to/model.pth"))*

- Loads the pretrained CNN weights from a .pth file.
- Make sure to match the exact path.



#### 4. Model Export to ONNX

```
def export_model():
```

```
...
```

- This function exports your PyTorch model to the **ONNX format**, which is a universal model format supported by many platforms.
  - A dummy input (5, 3, 28, 28) is used just to simulate the structure during export.
- 



#### 5. Model Inspection using ONNX (currently incomplete and commented)

```
def inspect_model():
```

```
...
```

- Loads the exported ONNX model and attempts to run inference using it.
  - Uses OpenCV + PIL to preprocess the image.
  - You attempted to use **onnx-caffe2 backend**, but this is an outdated library (ONNXRuntime is preferred now).
- 



#### 6. Braille Prediction Function

```
def make_prediction(img_path):
```

```
...
```

This is the core part of your logic.

**Function Steps:**

1. **Load image.**
2. **Divide image horizontally into Braille cells** using its aspect ratio.
3. **For each segment:**
  - Resize to 28x28, normalize and convert to tensor.
  - Run prediction using the CNN model.

- Based on prediction class index, map to a specific character/symbol:
  - 0–25: 'a' to 'z'
  - 26–36: special characters like space, #, ., : \ - ; ? ! ~

### **Post-processing:**

The function **decodes modifiers**:

- #: Indicates numbers are coming.
  - ~: Indicates capital letters/words.
  - Logic to transform Braille sequences into final readable **English text**.
- 

## 7. Special Character Mapping

Here's a sample of mappings:

### **Index Character**

26	Space ()
27	# (Number indicator)
28	.
29	,
30	:
31	\
32	-
33	;
34	?
35	!
36	~ (Capital modifier)

---

## **Output:**

The final string combines all predictions into meaningful English sentences, adjusting:

- **Capital letters**
  - **Numbers**
  - **Spacing and punctuation**
- 

## **Summary**

You've built a complete **Braille OCR (Optical Character Recognition)** system using CNN and PyTorch that:

- *Classifies segmented Braille characters.*
  - *Decodes modifiers for numbers and capitals.*
  - *Converts them to readable English text.*
  - *Includes model exporting for deployment using ONNX.*
- 

## **Suggestions**

-  *Use onnxruntime instead of onnx\_caffe2 (it's more modern and supported).*
-  *Add torch.no\_grad() during inference for performance.*
-  *Automate image segmentation using object detection in future (YOLOv8 maybe? 😊).*

Would you like me to help with improving ONNX export and inference using onnxruntime or build a UI for this in Streamlit?

## **3.FOLDER:BRAILLE.PY INDICATES:**

### **CODE:**

```
from flask import Flask, render_template, request,jsonify
```

```
import time
import os, sys
import pygame
from gtts import gTTS

sys.path.insert(0, 'final_model')
from final_model.space_recognition_original import
make_prediction

app = Flask(__name__)

@app.route("/")
def index():
    return render_template('home.html')

@app.route("/upload")
def upload():
    return render_template('upload.html')

@app.route("/upload_results", methods=['GET', 'POST'])
def save():

    import os
    # os.remove("output.mp3")
    # Save the image in the path
```



```
def play_audio(filename='output.mp3'):  
    # Initialize pygame mixer  
    pygame.mixer.init()  
    # Load the audio file  
    pygame.mixer.music.load(filename)  
    # Play the audio file  
    pygame.mixer.music.play()  
    # Wait for the audio to finish playing  
    while pygame.mixer.music.get_busy():  
        pygame.time.Clock().tick(10)  
  
@app.route("/upload_results_out", methods=['GET', 'POST'])  
def upload_results_out():  
    # Save the image in the path  
    # if request.method == 'POST' and 'fileField' in request.files:  
    s = request.files['img']  
    filename = s.save(r"C:\New folder\Braille\static\a.jpg")  
  
    img_path = r"C:\New folder\Braille\static\a.jpg"  
    predicted_letter = make_prediction(img_path)  
    print(predicted_letter, "jo")  
    # text_to_speech(predicted_letter)  
    # play_audio()
```

```
return jsonify(status="ok" ,result=predicted_letter)
```

```
if __name__ == "__main__":  
    app.run(debug=True, port=8001, host='0.0.0.0')
```

*DESCRIBES THE FOLLOWING:*

*Hi Nandana 🤙! Here's a detailed, human-understandable explanation of your **Flask-based Braille Text-to-Speech Web Application** — step-by-step. This app allows users to **upload a Braille image**, get the **decoded English text**, and **listen to it using gTTS + pygame**.*

---

## **FILE STRUCTURE (Assumed)**

*Your Flask project seems structured like this:*

```
Braille/  
|  
|——final_model/  
|   |——space_recognition_original.py ← your CNN +  
|   |——make_prediction function  
|——static/  
|   |——a.jpg           ← uploaded image
```

```
| └─ audio/
|   └─ *.mp3           ← generated speech files
└─ templates/
  ├─ home.html
  ├─ upload.html
  └─ display.html
└─ app.py             ← this Flask app
```

---

## **Code Overview**

### **Importing Libraries**

```
from flask import Flask, render_template, request, jsonify
import pygame, time, os, sys
from gtts import gTTS
```

- *Flask: Web app framework*
  - *pygame: For audio playback*
  - *gTTS: Google Text-to-Speech for converting Braille text to audio*
- 

## **Flask Routes**

### **/ → Home Page**

```
@app.route("/")
```

```
def index():
```

```
return render_template('home.html')
```

- *Renders home.html template.*
- 

### **/upload → Upload Page**

```
@app.route("/upload")
```

```
def upload():
```

```
return render_template('upload.html')
```

- *Renders upload.html with an upload form.*
- 

### **/upload\_results → Process Image Upload & Speak**

```
@app.route("/upload_results", methods=['GET', 'POST'])
```

```
def save():
```

```
s = request.files['fileField']
```

```
s.save(r"C:\New folder\Braille\static\a.jpg")
```

```
img_path = r"C:\New folder\Braille\static\a.jpg"
```

```
predicted_letter = make_prediction(img_path)
```

```
filename = text_to_speech(predicted_letter)
```

```
play_audio(filename)
```

```
return render_template('display.html', filename=filename,  
letter=predicted_letter)
```

### **What it does:**

1. Gets uploaded image (*fileField*) and saves as *a.jpg*.
  2. Calls your Braille CNN's *make\_prediction()* to convert image → text.
  3. Converts text to speech using *gTTS()*, saves the audio.
  4. Plays the audio using pygame.
  5. Displays result in *display.html*.
- 

### **/upload\_results\_out → API-style Endpoint (No UI)**

```
@app.route("/upload_results_out", methods=['GET', 'POST'])  
def upload_results_out():  
    s = request.files['img']  
    s.save(r"C:\New folder\Braille\static\a.jpg")
```

```
img_path = r"C:\New folder\Braille\static\a.jpg"
```

```
predicted_letter = make_prediction(img_path)
```

```
return jsonify(status="ok", result=predicted_letter)
```

- Used for backend-only API response (e.g., from a mobile app or frontend JS).
-

## *Text to Speech Functions*

### ***text\_to\_speech(text)***

```
def text_to_speech(text, lang='en'):
```

```
...
```

```
filename = r"...\\static\\audio\\{timestamp}.mp3"
```

```
gTTS(text=text, lang=lang).save(filename)
```

```
return filename
```

- Converts the predicted text into a .mp3 using gTTS.
  - Saves with a **timestamped filename** to avoid conflicts.
- 

### ***play\_audio(filename)***

```
def play_audio(filename):
```

```
    pygame.mixer.init()
```

```
    pygame.mixer.music.load(filename)
```

```
    pygame.mixer.music.play()
```

```
    while pygame.mixer.music.get_busy():
```

```
        pygame.time.Clock().tick(10)
```

- Plays the generated .mp3 using pygame.
- 

## *Summary*

 **Inputs:** User uploads a Braille image.

 **Processing:**

- CNN predicts each Braille character.
- Decoder assembles English sentence.
- gTTS converts to speech.
- Pygame plays the audio.

### Outputs:

- Displays recognized text.
- Plays back spoken output.
- Optionally returns a JSON (via API route).

## Suggestions to Improve

Area	Suggestion
 Audio Playback	Consider using HTML5 <audio> tag in display.html for platform-independent playback instead of pygame.
 Deployment	Use relative paths and os.path.join() instead of hardcoding "C:\\New folder\\...". This helps in deployment.
 Testing	Add checks if uploaded file is an image (e.g., .jpg, .png).
 File Management	Auto-delete old images/audio to save space.

<b>Area</b>	<b>Suggestion</b>
 REST API	<i>Improve the /upload_results_out to handle errors, invalid formats.</i>

---

*If you'd like, I can:*

- *Help convert this into a **mobile app** using **Flutter** or **React Native + Flask API**.*
- *Help create a **Streamlit interface** (faster for demo).*
- *Write display.html or upload.html if you don't have those yet.*

*Would you like help with one of those next? 😊*